4

The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM (Revised)

Alan L. Cox and Robert J. Fowler

Technical Report 263
May 1989

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM (Revised)

Alan L. Cox
Robert J. Fowler

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 263

Saturday May 6, 1989

Abstract

PLATINUM is an operating system kernel with a novel memory management system for *Non-Uniform Memory Access* (NUMA) multiprocessor architectures. This memory management system implements a *coherent memory* abstraction. Coherent memory is uniformly accessible from all processors in the system. When used by applications coded with appropriate programming styles it appears to be nearly as fast as local physical memory and it reduces memory contention. Coherent memory makes programming NUMA multiprocessors easier for the user while attaining a level of performance comparable with hand-tuned programs.

This paper describes the design of the PLATINUM memory management system with emphasis on the implementation of coherent memory and the factors that affect its performance. We measure the cost of basic operations implementing coherent memory. We also measure the performance of a set of application programs running on PLATINUM. Finally, we comment on the interaction between architecture and the coherent memory system.

PLATINUM currently runs on the BBN Butterfly Plus (TM) Multiprocessor.

This work is supported in part by U. S. Army Engineering Topographic Laboratories research contract no. DACA 76-85-C-0001, in part by ONR research contract no. N00014-84-K-0655, and in part by NSF research grant no. CCR-8704492.

SEC Y CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 263	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert J. Fowler and Alan L. Cox		8. CONTRACT OR GRANT NUMBER(s) DACA76-85-C-0001 N00014-84-K-0655
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department 734 Computer Studies Bldg University of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS D. Adv. Res. Proj. Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE May 1989
		13. NUMBER OF PAGES 22
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Res. Information Systems Arlington. VA 22217	US Army ETL Fort Belvoir VA 22060	15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Multiprocessor, operating system, memory management, NUMA, caching, cache coherency

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

PLATINUM is an operating system kernel with a novel memory management system for NUMA multiprocessor architectures. This memory management system implements a coherent memory abstraction. Coherent memory is uniformly accessible from all processors in the system. When used by applications coded with appropriate programming styles it appears to be nearly as fast as local physical memory and it reduces memory contention. Coherent memory makes programming NUMA multiprocessors easier for the user while attainging a level of performance comparable with hand-tuned programs.

20. ABSTRACT (Continued)

This paper describes the design and implementation of the PLATINUM memory management system, emphasizing the coherent memory. We measure the cost of basic operations implementing the coherent memory. We also measure the performance of a set of application programs running on PLATINUM. Finally, we comment on the interaction between architecture and the coherent memory system.

PLATINUM currently runs on the BBN Butterfly Plus^{tm} Multiprocessor.

Errata
TheImplementation of a Coherent Memory Abstraction
on a NUMA Multiprocessor: Experiences with PLATINUM
(Revised)

Alan L. Cox and Robert J. Fowler.
The University of Rochester
Computer Science Department

Technical Report 263
Wednesday May 10, 1989

p. – "implementation can good parallel" becomes "implementation can attain good parallel".

p1 – "extending of a" becomes "extending a".

p2 – "including including" becomes "including".

p8 – "the the" becomes "the".

p8 – "If one is found, the bus error must be a coherent memory fault. Otherwise, ..." becomes "If one is found, and it is determined not to be a protection violation fault, then it is a coherent memory fault".

p9 – "contains the index of an entry" becomes "contains a pointer to an entry".

p17 – "intalled" becomes "installed".

p19 – "The Natasha programming language ..." becomes "We will use the Natasha programming language".

p19 – "withl Mach" becomes "with Mach"

1 The Need for Transparent Management of Non-Uniform Memory.

PLATINUM is an operating system kernel designed to be a platform for research on memory management systems for *Non-Uniform Memory Access* (NUMA) multiprocessor architectures, those in which the distributed, shareable memory of the machine can be referenced by any processor on the machine, but the cost of accessing a particular physical location varies with the distance between the processor and the memory module. The name "PLATINUM" is an acronym for "Platform for Investigating Non-Uniform Memory". It supports the experimental evaluation of a family of software implementations of a *coherent memory* abstraction on top of non-uniform physical memory architectures.

One can achieve impressive speedup due to parallelism on a NUMA multiprocessor, but unfortunately this can entail a considerable effort. Because remote memory references are an order of magnitude more expensive than local references and because remote references are subject to several forms of potential contention, the physical location of data is critical to performance. On the BBN Butterfly (TM) Parallel Processor, a popular and productive way to deal with the problem of data location is to avoid the question by using libraries [24] and languages [30] that support message passing. When using distributed NUMA memory directly, however, one has to deal with data locality. This programming of data locality is reminiscent of the explicit management of memory hierarchies using overlays: attaining performance can be non-intuitive and can depend upon dynamic properties of program execution; worse, it has to be done explicitly by every application programmer. Thus, a programmer can easily expend far more effort "programming the memory architecture" to attain low latency and low contention than solving his or her problem.

Our goal is to explore the possibility of achieving performance comparable to that of hand-tuned programs with a simple, easy-to-program shared-memory model. PLATINUM is an exercise in doing this transparently in an operating system kernel on top of an existing NUMA multiprocessor. PLATINUM assumes neither special architectural support nor extensive language-specific assistance from a compiler, although we believe them to be important in the long run. It is crucial to present users with a simple model of shared memory whose implementation can good parallel performance. It is our thesis that the key to attaining this goal is to provide an efficiently implemented, uniform, and coherent model of memory to the user.

PLATINUM's implementation of coherent memory replicates and migrates data to the processors using that data, thus creating the appearance that memory is uniformly and rapidly accessible. The protocol for controlling this data movement is derived by extending of a directory-based cache coherency algorithm using selective invalidation [10; 1; 4]. The extension accomodates the NUMA architecture by including an option of not replicating or migrating data to local memory on an access miss, rather using the underlying remote access mechanism. This in effect dynamically disables caching on a block-by-block basis. This is crucial because when write-shared data is modified at fine temporal and spatial granularities the overhead of executing a coherency protocol can be more expensive than not having caching at all. This effect can be especially bad with the large block sizes and overheads associated with software-assisted caching. This is a critical distinction between NUMA memory management in PLATINUM and the software caching of Li's Distributed Virtual Memory [26] or the software controlled caching of the VMP Multiprocessor [12; 11].

Measured performance on real applications is a far better indicator of the success of a system than predictions based on either simplified analytic models or trace driven simulation extrapolated on a few seconds of the execution of a very different system. For that reason, rather than produce a paper design of the coherent memory, we decided to produce a small experimental kernel, PLATINUM, on which we could run a wider variety of experiments, varying both the policies and mechanisms of the memory management system as well as testing it against a wide variety of programs and

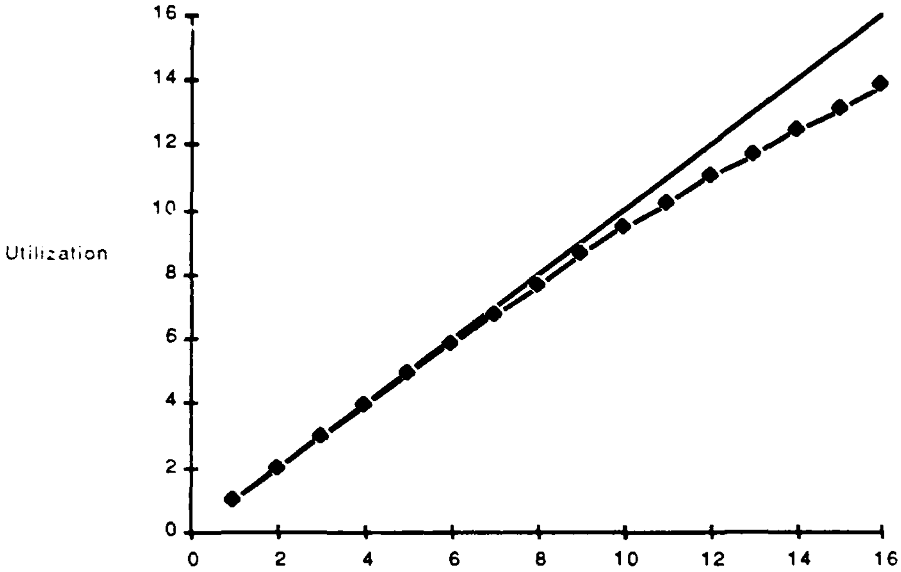

Figure 1: Gaussian Elimination Speedup

programming styles. Version 1 of PLATINUM runs on BBN Butterfly Plus (TM) Parallel Processors.

We are actively building a library of applications designed to test the performance of PLATINUM with a variety of programming styles that use different memory access patterns. The results are encouraging. Figure 1 plots the speedup of a program that simulates Gaussian elimination without pivoting on dense matrices. In this case the input is 800 by 800. This particular problem was chosen because it was used in performance studies of programming systems [15; 22] on earlier versions of the Butterfly. It simulates Gaussian elimination in the sense that it uses integer rather than floating-point operations, thus emphasizing the relative impact of memory latency with respect to the speed of arithmetic. Speedup on this program is limited by the serialization of block transfer operations at the memory module containing the pivot row of the current iteration.

There are several other test applications including including a family of merge sort programs and a simulator for recurrent neural networks that learn by backpropagation. This last program is especially interesting because its memory referencing behavior is inherently much less regular than that of the other programs, and because it was implemented an AI researcher rather than an operating system designer.

The design of PLATINUM targets factors such as ease of programming and performance, since these are the criteria by which the coherent memory abstractions should be judged. While other issues such as security, protection, and long-term storage have been considered in the abstract design, they have received only cursory attention in the current version.

1.1 Design Strategy

NUMA multiprocessor organization leads to memory management design choices that differ markedly from those that are common in systems designed for uniprocessors or UMA multiprocessors. If two or more processes on a uniprocessor are sharing read-only data such as a common code segment, it is wasteful to allocate multiple private copies. Such replication is expensive in terms of number of page frames used and in terms of the expense of copying the data. For example, to reduce this expense, Mach [27; 32] is designed to minimize the amount of data copying and replication through the use of copy-on-write and other techniques.

In contrast, extra data motion in the form of replication and migration can yield greatly improved performance on a NUMA machine. Placing data in the local memory of a processor that is using it decreases memory access latency. More importantly, a processor accessing local data is not performing remote operations that contend for remote memory modules and for the processor-memory switch. These two factors also motivate the use of caches in bus-based multiprocessors [18]. The advantages of replication and data motion distinguish the problem of managing memory on a NUMA machine from the same problem on uniprocessors and *Uniform Memory Access* (UMA) multiprocessors. Thus, the PLATINUM coherent memory subsystem attempts to replicate and migrate data to the processors using that data.

2 PLATINUM Overview

Since the goal is the exploration of transparent NUMA memory management, we chose to use familiar abstractions and interfaces as much as possible rather than develop new ones specifically for PLATINUM. This decision applied to both the interface presented to the user and some of the internal kernel interfaces. We chose Mach [5] as the prototype because of its separation of the virtual memory system into machine-dependent and -independent parts. The interfaces to PLATINUM were derived by stripping down the Mach model and then applying some further simplifications that were possible because the kernel runs on a single NUMA machine. Within this simplified model, PLATINUM coherent memory is implemented as a replacement for the machine-dependent part of the memory management system.

Given the initial successes with PLATINUM, its interfaces are being extended as required to provide added functionality and ease of programming to support larger experiments. We are also adding an instrumentation interface to the kernel to help interpret its behavior. The design is intended to make it easy to integrate PLATINUM coherent memory with Mach.

2.1 Abstractions

This section briefly describes the abstractions supported by the PLATINUM kernel. A comprehensive description of the interface can be found in [16].

The model that PLATINUM exports to user programs is a multiprocessor in which all primary memory accessible to user programs appears to be a fast (on average) shared physical memory module that is uniformly accessible from all of the processors in the system. The actual physical location of data in primary memory is hidden from the user. Page boundaries, however, are not hidden. PLATINUM allocates memory in page-aligned regions. This enables the user to reduce interprocessor interference by allocating shared data with different access patterns to different pages.

The fundamental abstractions supported by PLATINUM are the ***thread***, the ***memory object***, the ***port***, and the ***address space***. These objects all appear in a single flat global name space.

A ***memory object*** is an abstraction of an ordered list of memory pages. A range of pages within a memory object may be bound to any contiguous page-aligned virtual address range of the same size, subject to hardware alignment restrictions. Neither the virtual address range nor the access rights need be the same in every address space. Since they have global names, memory objects are the natural unit of data- or code-sharing between address spaces.

A *thread* is a kernel-schedulable thread of control. At any time it is bound to a single processor. An explicit migration operation can move it to another location. It is, however, constrained to execute within a single address space.

An *address space* is a list of bindings of memory objects and access rights to virtual address ranges. It defines the environment in which one or more threads may execute. The threads in a single address space may be distributed to multiple processors.

A *port* is a protected message queue that can have any number of senders and receivers. Messages are variable-length arrays of zero or more bytes. Globally named, ports provide a communication medium usable by threads that do not share access to a common memory object. Because **receive** operations on ports can block in the kernel, they also serve as the blocking synchronization mechanism by threads that do share memory.

Logical concurrency is realized through the use of multiple threads to implement a single application. True parallelism is realized by running those threads on multiple processors. Many different styles of communication and synchronization can be utilized by a collection of cooperating threads under PLATINUM. Communication between threads can be based on either either or both shared memory or message-passing via ports. Threads that coexist within a single address space share all of the memory objects mapped into that address space. This implies, in addition to data coherency, that these threads share a coherent view of the mappings of memory objects that constitute the shared space. A more restricted form of sharing is realized by mapping a shared memory object into multiple address spaces. The shared object can be accessed by all of the threads in those spaces, but the non-shared objects in each address space are protected from threads in other spaces.

3 Organization of the Memory Management System

A typical virtual memory system has traditionally managed a memory hierarchy consisting of a cache, a uniformly accessible primary memory, and a significantly slower secondary memory. The existence of remote primary memory on a NUMA multiprocessor adds at least one more level to this hierarchy. The PLATINUM memory management system is structured so as to separate the usual responsibilities of virtual memory management from the additional requirements imposed by the NUMA architecture. The memory manager is constructed in three layers. The highest layer is the *Virtual Memory* system. The middle layer is the *Coherent Memory* system. The lowest layer is the *Physical Map* system.

3.1 Virtual Memory System

The virtual memory system manages the mappings from virtual address ranges to memory objects and memory objects to coherent pages. In the "NUMA-Logical Map" on the left side of Figure 2 there are two address spaces into which three objects are mapped. Of the three objects, two are completely resident in primary memory and are therefore backed with coherent pages, while only a window of the third is resident.

The machine-independent part of the Mach virtual memory system is the prototype for this layer.

3.2 Coherent Memory System

The coherent memory system is responsible for the mappings from coherent pages to physical pages. These may be one-to-many. The left side of Figure 2 shows coherent-to-physical mappings for one of the three memory objects. When a processor accesses a coherent page for which it has no physical mapping the coherent memory system creates one to an appropriate physical page from the set

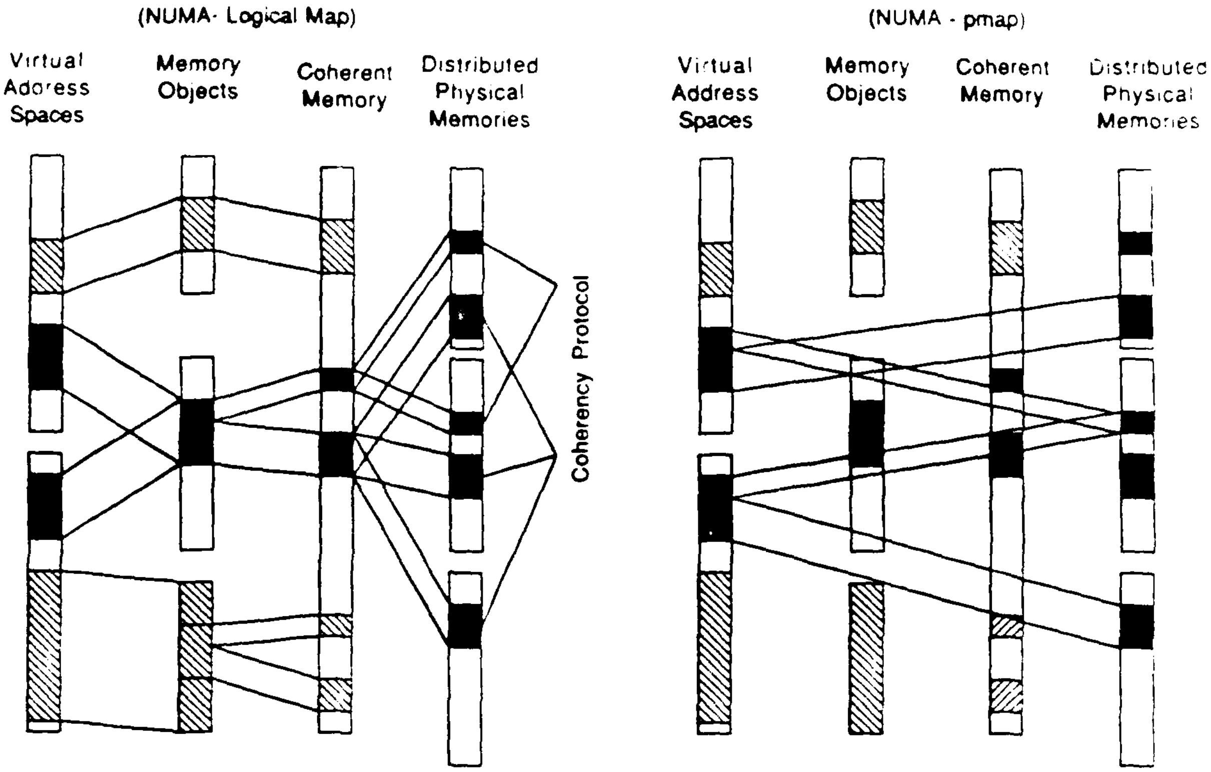

Figure 2: Logical and Physical Page Mappings

backing the coherent page. If the set contains no appropriate physical page, as determined by the replication policy, then the set will be changed by adding a new element and perhaps deleting others.

The coherent memory system also guarantees the consistency of the physical pages backing a coherent page.

Coherent memory is implemented by extending a directory-based protocol that performs selective invalidation to maintain coherency [10; 2]. For each coherent page the system maintains a directory of all physical pages backing it. A new physical page is added to the directory when the system chooses to replicate the coherent page. When a processor writes to a coherent page all but a single physical copy are invalidated and removed from the directory.

The protocol makes heavy use of the hardware memory management unit, a Motorola MC68851 on the Butterfly Plus, by restricting the access rights to physical pages in order to ensure the generation of bus errors by memory accesses which require action. Most transitions in the protocol are thus initiated by page faults and are performed by the bus error handler.

The coherent memory system consists of two main modules:

- *Cmap*
  The coherent map system is responsible for maintaining the coherency of the mappings from virtual to physical pages for each processor. It manipulates the address translation caches (ATC) of the memory management unit as well as the page tables. There is a separate local page table (Pmap) for each processor using an address space. The Mach pmap interface was used as a prototype for the Cmap interface to the virtual memory system.

- *Cpage*
  The coherent page system is responsible for allocating, freeing, and maintaining the coherency

of the data in the physical pages backing a coherent page. The Cpage system includes the coherent page fault handler and the *defrost daemon*. It also implements the policy that controls the replication of coherent pages. An interface is given to the virtual memory system for the allocation of coherent pages

3.3 Physical Map System

The coherent memory system depends on a simple machine-dependent page table and address translation cache management module. For each address space a physical page map (Pmap) is used to cache the composition of the logical mappings maintained by the of the virtual memory system and coherent memory system. Each physical mapping illustrated on the right side of Figure 2 is the composition of a corresponding sequence of mappings on the left side of the figure.

4 Design and Implementation

NUMA multiprocessor architectures are interesting because they offer a promise of scaleable parallelism using a shared memory model of computation. The key to achieving performance is decentralizing computations, maximizing concurrency, and minimizing contention for shared resources. It is especially important that the operating system kernel use this principle to avoid limiting the scaleability of the system. The design and implementation of the memory management system is based on three guiding strategies:

- Kernel operations and data structures should be decentralized
- Local data structures should be used whenever possible to minimize memory access latency and reduce memory module contention.
- The residual impact of contention between concurrent kernel operations should kept small.

These design principles led to specific implementation techniques. Wherever possible, atomic memory operations are used to implement concurrent data structures. If an explicit lock is necessary to implement a critical section, then its scope is minimized. Remote memory accesses in critical sections are avoided, especially within the coherent page fault handler. In some cases the algorithms and data structures have been designed so as to perform a modest number of local memory accesses rather than a single remote memory access. Replication is used extensively. When it is not possible to replicate data in the layers that implement coherent memory, it is scattered among the memory modules to reduce contention. Because the upper layers of the kernel are implemented on top of coherent memory, a large part of the replication is handled automatically.

The kernel address space consists of two regions, one in physical memory and the other in coherent memory. Kernel code and the data structures for the lowest layers of the kernel are in physical memory. Kernel code and read-only data are replicated, but writeable data can only have one copy. It is mapped for remote access by all but its local processor. The physical and coherent memory systems as well as physical device handlers must have their data structures in physical memory. The upper layers, however, have their data structures in the coherent memory region. These include the implementations of virtual memory, threads, and ports.

The kernel stacks, which are in the coherent memory region, require special handling. Otherwise, after movement of a thread between processors, the first attempt to access the kernel stack would generate a coherent memory fault, causing the processor to try to save its state on that same kernel stack. This circular dependence is broken by migrating the kernel stack explicitly before the corresponding thread is allowed to be activated at its new location.

The coherent and physical memory management systems use the following data structures:

- *Cmap*
  The mappings from virtual addresses to memory objects and from memory objects to coherent pages are maintained by the virtual memory system. For each address space the coherent memory system caches the composition of these mappings in a Coherent Map (Cmap). A Cmap contains a table of virtual-to-coherent page mappings (Cmap entries), a queue of recent changes to the address space, a bit mask denoting processors with this address space active and a Pmap for each of these processors. The message queue is used to maintain address space coherency.

- *Cmap entry*
  A Coherent map entry is analogous to a page table entry. It contains a pointer to the coherent page, an access control field, and a bit vector called the *reference mask*. If a processor has a virtual-to-physical translation for the coherent page in its Pmap, the bit corresponding to that processor is set.

- *Cmap message*
  Coherent map messages describe changes made to the virtual address space that affect virtual-to-physical mappings held by two or more processors. A message includes the virtual address and a directive either to invalidate the current translation or to restrict the access rights in it. Each processor is responsible for making the changes that affect it before it allows any thread in that address space to run.

- *Cpage table*
  The Coherent page table is the list of all available coherent pages. The number of coherent pages is determined by the amount of available physical memory. Each entry in the Cpage table describes the state of a Cpage. This information includes a directory of physical pages backing the Cpage and indicates whether there is a virtual-to-physical translation allowing write access. The directory consists of a bit mask indicating which memory modules contain a physical page backing the Cpage and a list of these physical pages. An entry also records the time of the most recent invalidation and whether the Cpage has been frozen by the replication policy.

- *Inverted Page Table*
  Each memory module contains an inverted page table which describes the state of each physical page in the module. An entry indicates whether the physical page is allocated and to which coherent page.

5 Shared-Memory Coherency

The shared-memory coherency problem has two major facets, data coherency and address space coherency. Much of the literature on coherent multiprocessor caches concerns the data coherency part of the problem. On UMA multiprocessors with coherent caches the address space coherency problem is primarily one of maintaining the consistency of address translation caches [8]. Given the lack of any direct haraware support for either form of coherency, PLATINUM solves both aspects of the problem in one unified framework; data coherency is implemented using a mechanism for *maintaining address space coherency.*

5.1 Coherency Protocol

To maintain shared memory coherency we extended a directory-based cache coherency protocol using selective invalidation [10, 2]. On a NUMA multiprocessor a physical page of memory need not be local to the node that accesses it. Thus, when a processor tries to access a coherent page that has no local physical page backing it, the PLATINUM coherent memory mechanism can always choose either to make a local copy of a page or to create a mapping to an existing remote page. In principle, the choice should be made by evaluating the tradeoff between the performance benefits of local access and the overhead of executing the protocol. The ability to use remote mappings is especially important when multiple processors make frequent, interleaved, and fine-grain modifications to a shared data structure. The resulting interprocessor interference causes the frequent execution of any protocol to maintain coherence among multiple copies. By using remote mappings the PLATINUM protocol can, in effect, selectively and dynamically disable replication and migration when interference is detected.

A coherent page can be in one of the following four states:

empty means that there are no physical pages backing the coherent page. Thus, there are no virtual-to-physical mappings to this page. Only unaccessed zero-fill memory can be in this state.

present1 means that there is exactly one physical page backing the coherent page and all virtual-to-physical mappings are restricted to read access. A virtual-to-coherent mapping may permit write access to the coherent page, but the virtual to physical mapping is restricted in order to implement the coherency protocol.

present+ means that there are two or more physical pages in different memory modules backing the coherent page. All virtual-to-physical mappings for the coherent page are restricted to read access. As above, a virtual-to-coherent mapping may permit write access to the coherent page.

modified means that there is one physical page backing the coherent page and at least one virtual-to-physical mapping allows write access.

The **present1** state is distinguished from the **present+** state for performance reasons. The transition from **present+** to **modified** on a write miss requires the the invalidation of at least one virtual-to-physical mapping and the reclamation of at least one physical page. The transition from **present1** to **modified** requires neither.

Most transitions between states are triggered by bus errors. When a bus error occurs during an attempted access to a non-**empty** Cpage, the Cpage system can either map an existing physical copy for remote access, or create and then map a local physical copy. For example, if there is a write miss on a Cpage in the **modified** state, the choice is between mapping the existing physical copy or allocating a local physical page, copying the data, and then invalidating the original copy. Similar decisions arise for the other cases.

A policy module within the Cpage system chooses the appropriate action on each bus error. The current policy is driven by the Cpage's history of recent invalidations. Cpages that have not been recently invalidated are replicated. On the other hand, recent invalidation is used as an indication that the Cpage is being shared by processors that are writing to it. The Cpage system uses this information to limit the overhead of running the protocol.

5.2 Implementing Data Coherency

Data coherency is implemented by the bus error fault handler. This fault handler is split across the coherent and virtual memory layers. When a bus error occurs, the Cpage fault handler searches the Cmap for an entry that maps the faulting virtual address. If one is found, the bus error must be a coherent memory fault. Otherwise, the fault is passed to the virtual memory fault handler.

The Cmap entry contains the index of an entry in the Cpage table. The fault handler tests the bit mask in the Cpage to discover whether a local physical page is backing it. Since Cpages may be shared by multiple address spaces a local physical copy may already exist. If a local copy exists, the handler applies a hash function to the index of the Cpage and scans the inverted page table to find the physical page. The inverted page table is used rather than the list of physical pages in the Cpage's directory because the former is guaranteed to use strictly local memory accesses, thus decreasing both latency and potential contention. Even when contention is not a problem it is cheaper to scan over a few collisions in the inverted page table than to search the list of physical pages with remote memory accesses.

If there is no local physical copy and the fault is a read miss, the fault handler consults the replication policy module to determine whether or not it should replicate the Cpage. If the Cpage is to be replicated, the handler uses the inverted page table to find a free physical page. It then allocates the physical page by entering the address of the Cpage entry in the inverted page table entry for the physical page. If the existing state of the Cpage is **modified**, the handler uses the address space coherency mechanism to restrict all virtual-to-physical translations for the Cpage to read-only access. The handler then performs a block transfer from another physical copy, and adds the physical page to the directory.

If the policy indicates that the Cpage should be *frozen* rather than replicated, there can only be one physical page backing the Cpage. Furthermore, the Cpage must be in a **modified** state. The handler simply maps the remote physical page for the access access rights allowed by the virtual memory system.

Similar sequences of actions occur on a write miss. For example, if the state of the Cpage is **present+**, the handler first uses the address space coherency mechanism to invalidate all virtual-to-physical translations for the remote physical copies, and then frees all of these pages. The handler concludes by mapping the chosen physical copy of the Cpage with the necessary access rights.

5.3 Implementing Address Space Coherency

When an address space is modified by the addition of new mappings or by relaxing the protection on a range of virtual addresses, it is easy to distribute the changes. Any processor attempting to use its expanded privilege will cause a bus error and thus be able to discover and react to the change. On the other hand, when an address space is restricted by removing mappings or increasing protection, some additional mechanism is necessary to ensure consistency. For example, consider a UMA multiprocessor with a single shared page table per address space. Since page table entries are cached in the address translation cache (ATC) of each processor's hardware memory management unit, these cached copies must be invalidated whenever the corresponding page table entry is invalidated or restricted. Because address translation caches are usually private to the processor to which the MMU is attached, multiprocessor operating systems such as Mach implement this part of the address space coherency protocol with a software shootdown mechanism [8]. The PLATINUM address space coherency mechanism is very different from that used in Mach. The differences arise largely because the PLATINUM mechanism was designed specifically for NUMA multiprocessors.

Because code and data are replicated in PLATINUM, each processor needs to have its own private set of virtual-to-physical mappings for each address space. While Mach uses a single shared page table (Pmap) per address space, each processor in PLATINUM must have its own private Pmap per address space. Since a Pmap is only a cache of the valid virtual-to-physical translations, it need not contain mappings for everything in an address space, rather only a working set for that processor. Thus, in contrast with a scheme examined by Holliday [19], scaleability is not restricted by replication of page tables.

In addition to reducing latency and contention, using a local, private Pmap for each processor allows the construction of a fast address coherency mechanism. Black *et al.* discuss two problems that result from multiple processors sharing a single Pmap in Mach. If the processor initiating the address space change instructs a target processor to flush its ATC before updating the Pmap, the target processor may reload an inconsistent entry. If, on the other hand, the initiating processor updates the Pmap before instructing the target processor to flush its ATC, the target processor may write back its ATC entry to update the reference or modify bits, thereby creating an inconsistent Pmap. Their solution to these problems is to stall the target processors while the initiator changes the Pmap. Since PLATINUM uses a Pmap per processor, it does not face either of these problems.

A consequence of the replication of mapping information is that the Pmaps must be kept coherent as well as the ATC's. Address space coherency and data coherency are thus unified by a single mechanism that maintains a consistent set of virtual to physical mappings for each coherent page. Part of the protocol is performed by the processor initiating the address space change and part is performed by the processors sharing the address space with the initiator. They communicate through the Cmap message queues and through interprocessor interrupts.

The initiating processor posts a short message describing the change to the Cmap message queue of each address space affected by it. A change to a specific address space affects only that space, but a change of mappings caused by executing the data coherency protocol must affect every address space in which the Cpage is mapped. Part of each message is the bit mask specifying the set of target processors that eventually have to apply the change to their Pmap for this address space. This set is exactly the set of processors appearing in the reference mask of the Cmap entry for this Cpage. The set of target processors is thus restricted to those that are actually using a mapping for this Cpage. Furthermore, a processor needs to be interrupted to perform the change only if the address space is currently active. The remainder of the target processors will update their Pmaps when they activate the address space. In contrast, the Mach TLB shootdown mechanism must interrupt each processor with the address space activated, even if that processor has never referenced the page.

On the target processors the updating is executed by a Cmap synchronization handler that is called either from an interprocessor interrupt or as part of the activation of an address space. The handler scans the queue of change messages. If the processor appears in the target mask of a message, it applies the change to its Pmap and removes itself from the target mask. The last target processor to see a message removes it from the list.

The memory management system obtains a significant reduction of overhead by deactivating the kernel address space when a processor begins running in user mode. This reduces the number of inter-processor interrupts each processor receives. When a processor reenters the kernel to service a trap or interrupt, it has to reactivate the kernel address space before it can access coherent memory. Furthermore, kernel code that runs at the interprocessor interrupt level or higher is not allowed to access coherent memory.

6 Performance Measurements and Analysis

All of the performance measurements were gathered on a 16-processor BBN Butterfly Plus Multiprocessor. A processing node on this machine consists of a 16.67MHz MC68020 with a MC68851 MMU and 4 MBytes of physical memory.

6.1 Basic Operation Performance

The current page size is 4K bytes. The copying of data in a PLATINUM page migration operation is a kernel-initiated, page-aligned block transfer of a known size. This can be done in 1.11 ms, somewhat faster than can be achieved with the more general block transfer made available to users.

The total time to execute the protocol for a read miss that replicates a non-**modified** page ranges from 1.34 ms to 1.38 ms. The shorter time corresponds to the case in which the relevant kernel data structures are local, while the longer time corresponds to all remote accesses. Of this time, data copying accounts for 1.11 ms and the fixed overhead of allocating and mapping a physical page accounts for the remaining 0.23 ms to 0.27 ms.

The total time for a read miss that replicates a **modified** page ranges from 1.38 ms to 1.59 ms, if only one processor has to be interrupted to restrict its mapping to read-only access. The fixed overhead in this case ranges from 0.27 ms to 0.48 ms. The additional cost compared to a read miss on a non-**modified** is due to the address space coherency protocol.

The total time for a write miss on a **present+** page ranges from 0.25 ms to 0.45 ms in the case in which only one processor has be interrupted to invalidate its mapping and one physical page freed. For up to 16 processors, the incremental cost of interrupting each additional processor and freeing a physical page is no more than 17 μs. Freeing a physical page uses one remote memory read and and one write, accounting for about 10 μs of this time. We therefore believe that the incremental cost of interrupting a processor to restrict a mapping to be about 7 μs. In contrast, Black *et al.* report an incremental cost of 55 μs on a 16-processor NS32332 Encore Multimax [8].

6.2 When does it pay to migrate a page?

To decide when it is appropriate to migrate a data page rather than make a remote mapping, it is necessary to estimate the relative costs of each of these options. The following analysis is based on the contention-free costs of remote memory access. Contention, both at the memories and in the switch, increases latency by serializing requests. In the presence of contention the benefits of replication can be much higher than indicated here.

Suppose a data structure, X, is shared and written by p processors; further suppose that X is the sole occupant of a coherent page. Each processor operates on X in a critical section as follows: obtain the lock for X, perform a computation f entailing r memory references on it, and release the lock. If this operation were encapsulated in a procedure call it might be performed in one of three ways:

- The operation might be executed on the processor requesting the operation and leave the data where it is. The operation is an ordinary procedure call and access to the data can be any combination of local and remote memory references.
- The data and the process executing the operation might be co-located by moving the data to the processor requesting the operation. The operation is an ordinary procedure call, and access to the data uses local memory references.
- The data and the process executing the operation might be co-located by performing a remote procedure call. Access to the data uses local memory references.

While implementations of languages such as Emerald [21] and Natasha [13] on top of PLATINUM would utilize the third option, the analysis addresses the choice between the first two. Let C_{remote} be the cost of performing the operation using remote memory references, C_{local} be the cost of using local references, and $C_{migrate}$ be the cost of moving the data. It is always cheaper to move the data when

$$C_{remote} > g(p)C_{migrate} + C_{local}, \qquad (1)$$

where $g(p)$ is a function that expresses the number of data movements necessary to save a remote operation. It is the ratio of the total number of executions of f to the number of executions of f performed using remote data operations when the data is not moved.

When p processors access X in strict round-robin order, $g(p) = p/(p-1)$. For example, consider two processors that alternate in touching X. If X is not moved, there will be one remote and one local execution of f per cycle. If it is moved, there will be two local executions of f and two data movements per cycle. Thus $g(2) = 2$. This is the worst-case scenario for application behavior.

Let

T_l be the time to perform a typical local memory reference on a 32-bit word. On a Butterfly Plus this is about 320 ns.

T_r be the time to perform the corresponding remote memory reference. On the Butterfly Plus this is about 5000 ns to read a 32-bit quantity. Write operations are faster.

T_b be the time to copy a word in a page migration operation. This is about 1100 ns on the Butterfly Plus

s be the size of a page expressed in terms of the typical unit of access, on the Butterfly Plus a 32-bit word

Define $\gamma \equiv r/s$. For example, if the size of X is s and f reads and writes every location in X, $\gamma = 2$. On the other hand if X occupies only half the page and f writes one half of X's data, $\gamma = 0.25$.

We therefore have $C_{local} = \gamma s T_l$, and $C_{remote} = \gamma s T_r$. The cost of migration is divided into the cost of block transfer, sT_b, plus a fixed overhead, about .48 ms in the current implementation. Substituting these into equation 1 and rearranging the terms, we conclude that it always pays to migrate when

$$s > \frac{107g(p)}{\gamma - .24g(p)} \qquad (2)$$

Note that the constant in the numerator is proportional to the fixed overhead of the migration operation and that the coefficient of $g(p)$ in the denominator is the ratio $T_b/(T_r - T_l)$. From this we make the following observations:

- For determining when migration is economical, the ratio $T_b/(T_r - T_l)$ of block transfer time to the time that can be saved by using local rather than remote memory access operations is the single most important characteristic of the architecture. It puts a lower bound on the minimum access ratio γ for which migration makes sense for any block size. This in turn bounds the minimum useable page size. *The existence of a fast block transfer mechanism is vital to the success of any program that performs migration and replication!*
- For each γ, the need to amortize the fixed overhead of the coherence protocol puts a lower bound on the page size that can be used economically. For a fixed γ, a decrease in overhead results in a proportional decrease in the minimum page size for which migration makes sense.
- As the number of processors sharing X increases, $g(p)$ tends towards 1, thus making migration more attractive

These factors all determine the granularity of data access that must be seen in the application to ensure that migration is always the correct action for a given page size. Some values for equation 2 are presented in Table 1. For the current page size of 4K bytes, or 1K words, γ must be greater than $0.34g(p)$ for migration to be attractive if the reduction of latency in the absence of contention is the evaluation criterion.

This analysis emphasizes the importance of coarse data granularity for attaining good performance on a NUMA machine. A larger page size over which to amortize the fixed overhead allows us

γ	S_{min}, minimum page size	
	$g(p) = 1$	$g(p) = 2$
.24	never	never
.35	973 words	never
.48	435	never
.60	298	1784
.75	210	793
1.0	141	412
1.5	84	210
2.0	61	141

Table 1: Equation 2 evaluated at some interesting points. It always pays to migrate data when the page size is greater than S_{min}.

to tolerate a slightly smaller γ. For a fixed granularity of data access that is smaller than the size of a page, however, γ is inversely proportional to page size and therefore decreases too rapidly to take advantage of this effect. On the other hand, if a program has a granularity of data access that is greater than the size of a page, γ remains more or less constant as pages grow. Increasing page size in this case allows more effective amortization of the overhead.

6.3 Application Performance

The preliminary performance measurements were done on three application programs running on PLATINUM. Each of these programs has a memory access pattern distinct from the others. In addition to timing data, the kernel reports summary statistics pertaining to memory management. For each Cpage this includes the number of coherent memory faults, a measure of contention in the Cpage fault handler for that page, and whether the Cpage was frozen by the replication policy.

Gaussian Elimination

The first application we examined was the simulation of Gaussian elimination described in the introduction. This particular computation was chosen because it had been studied previously on an earlier version of the Butterfly for a variety of programming systems and styles [22; 23]. LeBlanc compared the performance of an implementation on the Uniform System from BBN [6] with Gaussian elimination implemented on SMP [24], a message passing library developed at the University of Rochester.

The PLATINUM version is most similar to a coarse-grain implementation on the Uniform System found to be the most efficient in LeBlanc's study. The Uniform System provides the user with lightweight threads of control and a globally-shared memory. Data in the globally-shared memory is scattered throughout the machine to reduce contention. To exploit the relative speed of local memory, user code typically copies shared data into private local memory using block transfer, accesses it tnere, and, if the data is modified, copies the new value back to the globally accessible location. Like the Uniform System implementation, the PLATINUM implementation uses a single thread running in a shared address space on each processor. We used the same 800x800 matrix.

The differences between the two versions of the Butterfly reduce the utility of quantitative comparisons of performance measures. Nevertheless, such measures provide a framework for qualitative comparison. The program running on PLATINUM yields a speedup of 13.5 versus 10.6 for the Uniform System program [22]. In contrast, the SMP message-passing implementation yielded a speedup of 15.3.

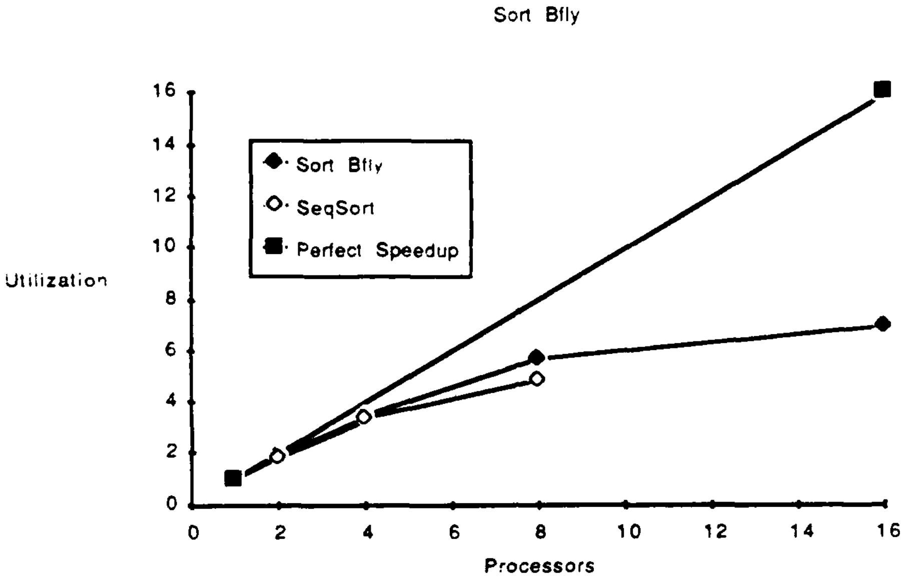

Figure 3: Merge Sort Speedup

The PLATINUM program is much shorter and simpler than the SMP implementation, which, in turn, is much shorter and simpler than the Uniform System implementation. The transparent caching performed by the coherent memory system eliminates the need for the programmer to explicitly manage the location of data structures. A crucial performance advantage of the PLATINUM implementation over the Uniform System program is that rows of the matrix do not have to be copied back to the globally-shared memory to make them accessible to other threads of the program.

An examination of the post-mortem statistics gathered by the kernel shows that the PLATINUM implementation exhibits high contention in the Cpage fault handler for certain Cpages. This is attributable to the replication of the data backing a Cpage when it contains the pivot row. As expected, the memory manager froze the Cpage containing an array of event counts used for synchronization.

Merge Sort

Another program used for measurements is a parallel merge sort that uses a tree of merge operations, each of which is performed by a single thread. We chose this program because it had been studied on a Sequent Symmetry Multiprocessor [3]. The Sequent Symmetry is a UMA multiprocessor. The one used in the study had model A processors with 8Kbyte write-through caches.

Figure 3 shows the measured speedup curves for this program. The program shows better speedup running on the Butterfly Plus under PLATINUM than on the Sequent Symmetry for the same size problem on the same number of processors. We believe this is due to the small cache size and write-through policy on the Sequent. During each merge phase one half of the data to be merged will already be in the merging processor's local memory. Furthermore, with the linear access pattern of merging, the processor will touch all of the data prefetched by each coherent page fault. The problem is large enough, however, that none of the data will remain in the Sequent cache between merge phases.

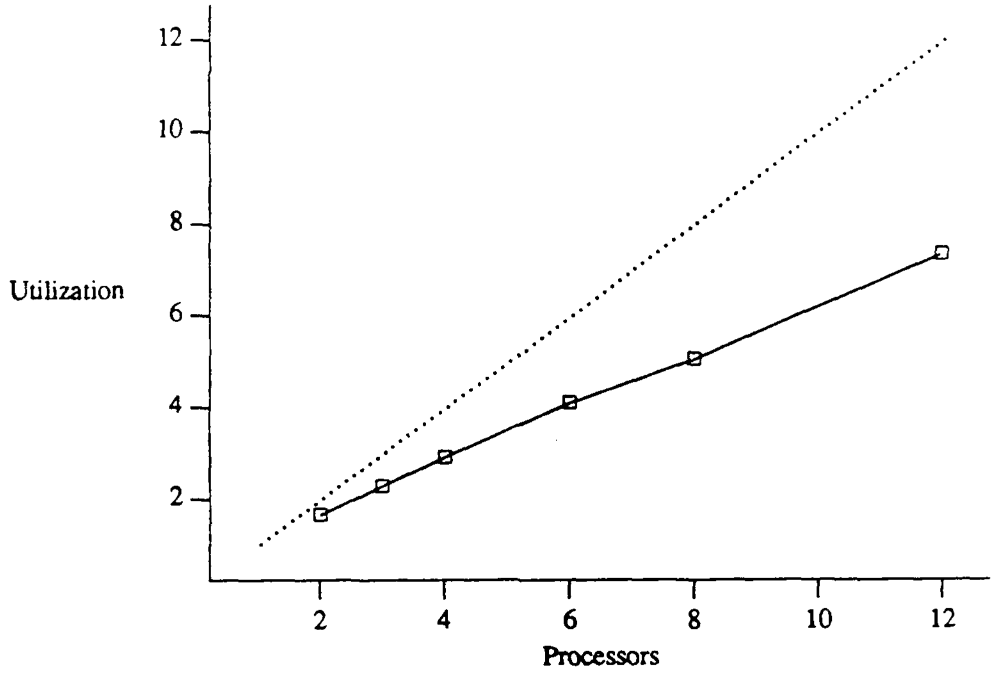

Figure 4: Recurrent Backpropagation Simulator Speedup

Connectionist Simulator

The third application is a simulator used by neural network researchers at the University of Rochester studying recurrent backpropagation networks [31]. Unlike the others, this program was developed by someone with no previous experience programming the Butterfly Plus. While the other programs were written to exploit coarse-grain parallelism on large amounts of data, the simulator operates on much less data and at a finer granularity. The backpropagation algorithm [28] works by propagating signals from an input to an output vector through successive layers of units. When the signals reach the output layer, they are compared with a target output and error signals are propagated backward to the input through each of the layers. These error signals are used to modify the strength of the connections between units so that the network eventually learns how to get the correct output for each of the inputs. An interesting approximation of the backpropagation algorithm is obtained by running the algorithm on a parallel machine without synchronizing the computation into rounds.

We measured the performance of a simulation of a network learning a classic encoder problem [28]. There are three layers (16-8-16) of units and 16 pairs of inputs and outputs. Each unit reads from each of the units in the previous layer a vector of 16 activations (corresponding to each of the patterns) and from each unit of the next layer a vector of 16 error signals. After multiplying the activations and error signals by the correct weight, the unit computes and updates its own activation and error signal. The task is terminated when the total error (difference between the targets and the computed outputs) falls below a fixed threshold. The non-determinism produced by the lack of synchronization introduces negligible variability of execution time.

The simulator is parallelized by simple for-loop parallelization on units. Each processor is given responsibility for a random subset of the units. It continuously simulates these units accessing the activation and error vectors from the adjacent layer units. Figure 4 plots the speedup for the simulator.

7 Replication Control Policy

While the replication and migration of data can have significant benefit in many cases, the overhead of trying to maintain coherency in the presence of fine-grain write-sharing could be prohibitively expensive. In such circumstances it is less expensive to use remote memory access than to try to migrate or replicate the data. Since the choice between data movement and remote access depends upon the relative costs of the alternatives, we have delayed discussing the replication control policy until after presenting the details of the mechanism and of its performance. We describe the interim policy currently in use in PLATINUM. We are also experimenting with alternative replication control policies.

Since invalidations occur as a result of interprocessor interference, all policies use a recorded history of recent invalidations to estimate the interference for each coherent page. The current version uses a minimal history consisting of a timestamp for the most recent invalidation by the coherency protocol of a mapping for that coherent page. On a bus error handled by coherent memory, a coherent page is replicated or migrated if the the last invalidation by the protocol was at least t_1 in the past. If it was invalidated within t_1, the coherent page is frozen rather than replicated. Since invalidations cause the Cpage to go into the **modified** state and since it could not have been replicated since then, there can only be one physical page backing a Cpage that is frozen. That physical page is mapped for remote access. We tried two policies for dealing with subsequent faults occurring after the t_1 ms period expires. The default policy is to continue to create remote mappings for the Cpage until the page is explicitly thawed. The alternative is to allow the frozen coherent page to be replicated and thus thawed as a consequence of an attempted access. The programs we have examined thus far exhibit no significant difference in performance between these policies.

Based on the speed of the Butterfly processor and the need to amortize the replication of a coherent page over a reasonable number of accesses, t_1 is currently set to 10 ms. A few tests indicated that application performance is insensitive to varying t_1 from 10 ms up to about 100 ms. Once the collection of application programs has grown to a reasonable size we will perform systematic experiments on the effects of varying this and other parameters.

Once all the threads that will touch a page have mappings to a frozen page, further access to that coherent page cause neither additional bus errors or nor the associated overhead. Since the coherency protocol as described thus far is driven strictly by bus errors, the page could remain frozen permanently. While it may be appropriate to freeze a page at a particular point in the execution of a program, a change in the access pattern of that page may make it desirable to thaw it in the future. PLATINUM therefore has a simple mechanism for thawing pages, thus allowing the memory manager to react to phase changes as well to thaw any incorrectly frozen pages.

The Cpage module maintains a list of frozen Cpages and a clock interrupt every t_2 seconds activates the *defrost daemon* to invalidate all mappings to the frozen pages. Subsequent access attempts will cause faults that may replicate or migrate a recently thawed coherent page. To keep the overhead low t_2 is currently set to 1 second. Reducing t_2 may allow coherent pages accidentally frozen to be replicated sooner, but it just adds overhead for coherent pages that should remain frozen.

An alternative implementation is to maintain the list of frozen pages as a priority queue ordered by thaw time. This allows the daemon to run more often than every t_2 seconds. It also would allow t_2 to be set adaptively on a per-page basis. Although there is evidence (see below) that thawing frozen pages is important for performance, we do not yet have reason to believe that a more sophisticated policy for thawing will have much effect. Since a more sophisticated policy would add overhead to the system, we plan to continue to use the simple policy described above until the problem is better understood.

A possible reason for the access pattern for a page to change is that two or more variables with different access patterns are in that page. For example, co-locating a synchronization variable such as a lock or event count with a read-only variable on one page can lead to problems because they demand very different treatments from the memory manager. Active use of synchronization variables will cause their pages to be frozen while a read-only variable should be replicable. The preferred solution to this problem is for the programmer, the compiler, and the run-time library for the language to be intelligent about the allocation of variables to virtual pages. Even if this allocation is not done well, however, it is still possible to improve performance in cases in which the variables are each used frequently only in different phases of the program.

Experiences with our first version of the Gaussian elimination program provide anecdotal evidence of the importance of intelligent memory allocation, thawing, and performance instrumentation. A variable whose value is the matrix size is written during a startup phase and is read-only for the rest of the execution. It is vital that each processor have a local copy of the matrix size since it is used in the termination test of the inner loop of the algorithm. Since we expected this variable to be replicated, the slave threads do not make private copies of this variable. A synchronization-flag variable was added later to facilitate interpreting execution times. It is used for barrier synchronization at the start of the elimination phase of the program and is not touched thereafter. Each slave thread spins waiting for the flag to be written by the master thread. This behavior freezes the Cpage, which also contains the matrix-size variable. This program was first run on a partially functional version of PLATINUM that did not yet have thawing intalled. In going from one processor to two the execution time *increased* because one thread had to access the matrix-size variable remotely in its inner loop. With four processors the execution time was less than with one processor, but with five or more processors the shared variable becomes a bottleneck and execution time rises slowly with number of processors.

Given the kernel's report on the behavior of the coherent memory system it was a simple matter to diagnose the problem and program around it using the simple expedient of having each thread keep a private matrix-size variable. Once thawing was installed, the old version of the program tooks less than two seconds more to run than the new version. The overhead of running the defrost daemon adds no measurable overhead to the new version of the program.

8 Experiences Programming on a Coherent Memory

In our experience, it is much easier to write applications to run on coherent memory than to run on non-uniform physical memory. PLATINUM programs are smaller than both Uniform System programs and programs using message-passing styles because it is not necessary to generate code either for explicit communication, or for explicit management of memory locality. The PLATINUM programmer can concentrate more effort on the problem the application is intended to address.

Despite the apparent familiarity of PLATINUM's abstract machine model, a programmer still needs to understand and apply certain fundamental facts about parallel programming on a NUMA machine It is of overwhelming importance to avoid programming styles entailing fine-grain write-sharing. Whether memory is being managed automatically by the coherent memory manager or explicitly by the programmer, this fine-grain write-sharing introduces both latency that reduces the effective processor speed and memory contention that serializes logically parallel computations. Any parallelism at the processors is constrained by serialization at the memories. It is vital that most of the sharing of writeable data be done at coarse enough spatial and temporal granularities that a fast block transfer mechanism can be used effectively.

In order to allow the coherent memory manager a chance to effectively manage data locality, the programmer or compiler must be cognizant of the sharing properties of data. Data with different access patterns should not be co-located on a single page. The private data of each thread should be separated from private data of other threads and from shared data. Read-only data should be kept separate from modifiable data. Coarse-grain modifiable data should be separated from fine-grain modifiable data. A run-time library for defining disjoint memory allocation zones and for specifying page-aligned allocation helps PLATINUM programmers to do this with a minimum of effort, even without compiler support. Because a typical NUMA multiprocessor has a very large physical memory, the internal fragmentation introduced by this strategy has little impact and is vastly preferable to interprocessor interference.

9 Architectural Considerations

The benefits of replication cannot be measured solely in terms of the ratio of local to remote memory access times. As the degree of parallelism increases on a machine with a large number of processors, contention for memory modules and the interconnection network become the dominant factors determining performance. The most important impact of coherent memory is that it effectively uses local memories as caches to reduce contention.

An effective block transfer mechanism is critical to an efficient implementation of coherent memory. It should be both fast and asynchronous with respect to program execution. The analysis in Section 6.2 quantifies the importance of block transfer speed in one scenario. Although the Butterfly Plus has a fast, asynchronous block transfer mechanism, it consumes 75% of the available local memory bus bandwidth on both nodes involved in the transfer. Both processors are memory-starved during a block transfer. Redesigning the memory system to allow more concurrency between processing and block transfers would help to reduce further the effects of memory contention.

Although the Butterfly Plus does not have local data caches in the processor nodes, the PLATINUM coherent memory mechanism is compatible with a generation of NUMA multiprocessors with caches but without a hardware coherency protocol. Since all mappings are guaranteed to be private and local except for frozen pages, almost all data is cacheable. Replicating a **modified** page would, however, require flushing a write-back cache and thus slow the invalidation operation. Such local caches could be relatively cheap because they need not incorporate a hardware cache coherency protocol. In addition to reducing latency on local memory operations, local caches would reduce contention for the local memory module between the local processor and remote memory operations.

10 Related Work

The management of NUMA memory is a topic of considerable current interest. Recent studies of methods for managing the location of data in a NUMA machine include the analysis and simulation of competitively optimal NUMA memory management by Black *et al.* [7], Scheurich and DuBois' simulation of data migration in mesh-connected NUMA machines [29], and Holliday's simulation of data migration on a Butterfly [20]. The design of the Psyche memory manager [25] contains a layer that deals with NUMA data location issues. Bolosky has implemented NUMA memory management for Mach on the IBM ACE Multiprocessor Workstation [9].

While one effect of replication and migration in the PLATINUM coherent memory system is the reduction of latency, we contend that for large hardware configurations a far more important benefit is the reduction of memory and switch contention. Therefore, we have have not expended much effort attempting to design a mechanism for the optimal placement of frozen pages, those that are being actively modified at a fine granularity by multiple processors. While careful placement and migration can reduce average access latency in the absence of contention, there is no demonstrated reduction in contention. Since the proposed placement mechanisms are not cheap, entailing hardware reference counts [7; 29] or simulations of reference counting in software [20; 25], we believe that it is better to have a simple, low-overhead placement policy and to devote more resources to reducing contention by reducing the amount of fine-grain write-sharing.

11 Status and Future Directions

Our experiences thus far indicate that the PLATINUM memory management system will achieve its goals. Foremost, the memory management system makes it easier to program a NUMA architecture without an unacceptable sacrifice in performance. Although initial programming experiments used the kernel interface directly without too much programmer effort, we are rapidly accumulating run-time libraries, shells, and other support software to further ease the programming process. An important part of this will be the installation of instrumentation for performance monitoring, analysis, and visualization [17]. The feedback from such instrumentation is useful to both application programmers and implementors of compilers for NUMA machines.

We are continuing to study the behavior of the coherent memory system under a variety of applications. Once the collection of applications has grown to a reasonable size we will begin a series of experiments systematically varying the implementation by changing parameters such as page size and replication control policy.

The kernel itself is designed to scale well to machines with a much larger number of processors. Its decentralized design keeps the number of remote memory accesses in the kernel to a minimum. We are particularly pleased with the success of the decentralized and concurrent implementation of the coherency protocol, especially the low incremental cost per shootdown and the techniques for reducing the number of processors involved in a shootdown.

Although providing coherent memory transparently in the operating system has proven itself useful, it is not hard to construct scenarios in which better performance could be obtained if the memory manager had access to some hints and directives from the application program. The kernel interface will be extended to support these. While such information could be provided by the programmer directly, this additional burden runs contrary to the goal of providing a simple programming environment. We therefore anticipate that a programming language and its run-time support will utilize these hooks. The Natasha programming language for such experiments [14; 13]. The Natasha compiler is near completion and will generate code which can be executed on PLATINUM.

In its current incarnation, PLATINUM is a limited experimental platform for experimenting with the implementation of coherent memory. We will extend it as necessary to serve this purpose. On the other hand, dealing with issues such as file systems and protection are not in our plans. When and if it becomes appropriate to make coherent memory available in a general purpose operating system, we anticipate reintegrating those parts of PLATINUM withl Mach.

Acknowledgements.

The second author would especially like to thank Niki Fowler for her editorial assistance on the revised version of this paper.

References

[1] A. Agarwal and A. Gupta. Memory-reference characteristics of multiprocessor applications under Mach. *Performance Evaluation Review*, pages 215–225, May 1988.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 280–289, June 1988.

[3] R. J. Anderson. An experimental study of parallel merge sort. Technical Report 88-05-01, Department of Computer Science, University of Washington, May 1988.

[4] J. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors*. PhD thesis, Department of Computer Science, University of Washington, February 1987.

[5] R. Baron, D. Black, W. Bolosky, J. Chew, D. Golub, R. Rashid, A. Tevanian, and M. Young. *Mach Kernel Interface Manual*. Carnegie-Mellon University, 1988.

[6] BBN Laboratories, Cambridge, Massachusetts. *The Uniform System Approach To Programming the Butterfly Parallel Processor*, October 1985.

[7] D. Black, A. Gupta, and W.D. Weber. Competitive management of distributed shared memory. In *Spring Compcon*, 1989.

[8] D.L. Black, R.F. Rashid, D.B. Golub, C.R. Hill, and R.V. Baron. Translation lookaside buffer consistency: A software approach. Technical Report CMU-CS-88-201, Department of Computer Science, Carnegie-Mellon University, December 1988.

[9] William J. Bolosky, Michael L. Scott, and Robert P. Fitzgerald. Simple but effective techniques for NUMA memory management. Technical report, Department of Computer Science, University of Rochester, March 1989.

[10] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[11] D. Cheriton, A. Gupta, P. Boyle, and Hendrik Goosen. The VMP Multiprocessor: Initial experience, refinements and performance evaluation. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 410–421, June 1988

[12] D. Cheriton, G. Slavenburg, and P. Boyle. Software-controlled caches in the VMP Multiprocessor. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 366–374, June 1986.

[13] Lawrence A. Crowl. A uniform object model for parallel programming. In *Proceedings of the ACM SIGPLAN Workshop on Concurrent Object-Based Programming*, pages 25–27, September 1988. Appeared in ACM SIGPLAN Notices 24(4), April 1989.

[14] Lawrence A. Crowl. A uniform object model for parallel programming. Distributed at the 1988 Workshop on Object Based Concurrent Programming, September 1988.

[15] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance measurements on a 128-node Butterfly Parallel Processor. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 531–540, August 1985.

[16] R. Fowler and A. Cox. An overview of PLATINUM: A platform for investigating non-uniform memory. Technical Report TR-262, Computer Science Department, University of Rochester, November 1988.

[17] R.J. Fowler, T.J. LeBlanc, and J.M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. In *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distribute Debugging*, pages 74–182, Madison, Wisconsin, May 1988. Association for Computing Machinery.

[18] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, 1983.

[19] Mark A. Holliday. Page table management in local/remote architectures. Technical Report CS-1988-2, Department of Computer Science, Duke University, July 1988.

[20] Mark A. Holliday. Reference history, page size, and migration daemons in local/remote architectures. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 104–112, April 1989.

[21] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, pages 109–133, February 1988.

[22] T. J. LeBlanc. Shared memory versus message-passing in a tightly-coupled multiprocessor:a case study. Technical Report Butterfly Project Report 3, Computer Science Department, University of Rochester, January 1986. A shorter version appears in Proceedings of the 1986 International Conference on Parallel Processing, August 1986, pp. 463–466.

[23] Thomas J. LeBlanc. Problem decomposition and communication tradeoffs in a shared-memory multiprocessor. In Martin Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, volume 13 of *The IMA Volumes in Mathematics and its Applications*, pages 145–163. Springer-Verlag, 1988.

[24] Thomas J. LeBlanc. Structured Message Passing on a shared-memory multiprocessor. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 188–194, January 1988.

[25] Thomas J. LeBlanc, Brian D. Marsh, and Michael L. Scott. Memory management for large-scale NUMA multiprocessors. Technical report, Department of Computer Science, University of Rochester, March 1989.

[26] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, September 1986.

[27] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Parallel Distributed Processing*, chapter Learning internal representations by error propagation, pages 318–364. MIT Press, 1986.

[29] C. Scheurich and M. DuBois. Dynamic page migration in multiprocessors with distributed global memory. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 162–169, June 1988.

[30] Michael L. Scott and Alan L. Cox. An empirical study of message-passing overhead. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 242–249, September 1987.

[31] P. Simard, M. Ottaway, and D. Ballard. Analysis of recurrent backpropagation. In *Connectionist Summer School Proceedings*, 1988.

[32] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, Austin, TX, November 1987. ACM.